**High Performance Computing**

**Practical File**



**Name**: Ashutosh Kumar

**Roll No.**: 2019UCO1647

**Branch &** **Section**: COE - 2

**Semester**: 6

**Subject Code**: COCSC18

**1. Run a basic hello world program using pThreads**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <pthread.h>

int thread\_count;

void \*Hello(void \*rank);

int main(int argc, char \*argv[]){

    long thread;

    pthread\_t \*thread\_handles;

    thread\_count = strtol(argv[1], NULL, 10);

    thread\_handles = malloc(thread\_count \* sizeof(pthread\_t));

    for (thread = 0; thread < thread\_count; thread++)

        pthread\_create(&thread\_handles[thread], NULL, Hello, (void \*)thread);

    printf("Hello from the main thread\n");

    for (thread = 0; thread < thread\_count; thread++)

        pthread\_join(thread\_handles[thread], NULL);

    free(thread\_handles);

    return 0;

}

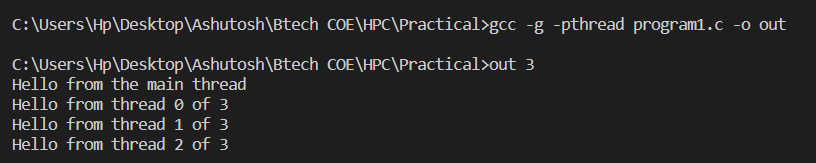
void \* Hello(void \*rank){

    long my\_rank = (long)rank;

    printf("Hello from thread %ld of %d\n", my\_rank, thread\_count);

    return NULL;

}



**2. Run a program to find the sum of all elements of an array using 2 processors**

#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

int main(int argc, char \*\*argv){

    MPI\_Init(NULL, NULL);

    int num\_procs;

    MPI\_Comm\_size(MPI\_COMM\_WORLD, &num\_procs);

    int rank;

    MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

    if (rank == 0){

        int n;

        printf("Enter number of elements : ");

        scanf("%d", &n);

        int arr[n];

        for (int i = 0; i < n; i++)

            arr[i] = rand() % 10000 + 1;

        printf("Array is -\n [ ");

        for (int i = 0; i < n; i++)

            printf("%d ", arr[i]);

        printf("]\n");

        int elem\_to\_send = n / 2;

        if (n % 2)

            elem\_to\_send++;

        MPI\_Send(&elem\_to\_send, 1, MPI\_INT, 1, 0, MPI\_COMM\_WORLD);

        MPI\_Send(&arr[n / 2], elem\_to\_send, MPI\_INT, 1, 1, MPI\_COMM\_WORLD);

        float t1 = clock();

        int local = 0;

        for (int i = 0; i < n / 2; i++)

            local = local + arr[i];

        int s\_rec = 0;

        float t2 = clock();

        printf("Time taken by process %d : %f\n", rank, (t2 - t1) / CLOCKS\_PER\_SEC);

        MPI\_Recv(&s\_rec, 1, MPI\_INT, 1, 2, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

        local = local + s\_rec;

        printf("Total sum of array is %d\n", local);

    }

    else{

        float t1 = clock();

        int size;

        MPI\_Recv(&size, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

        int arr[size];

        MPI\_Recv(arr, size, MPI\_INT, 0, 1, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

        float t2 = clock();

        printf("Total time for recieving : %f", (t2 - t1) / CLOCKS\_PER\_SEC);

        t1 = clock();

        int local = 0;

        for (int i = 0; i < size; i++)

            local = local + arr[i];

        printf("\nProcess %d sending sum %d back to main...\n", rank, local);

        t2 = clock();

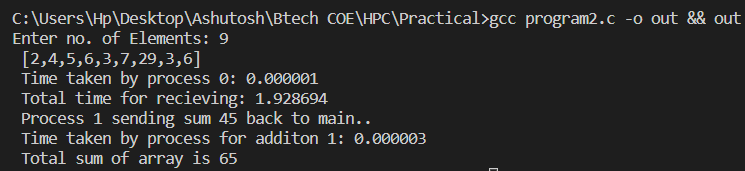
        printf("Time taken by process for addition %d : %f\n", rank, (t2 - t1) / CLOCKS\_PER\_SEC);

        MPI\_Send(&local, 1, MPI\_INT, 0, 2, MPI\_COMM\_WORLD);

    }

    MPI\_Finalize();

}



**3. Compute the sum of all the elements of an array using p processors**

#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

int main(int argc, char \*\*argv){

    MPI\_Init(NULL, NULL);

    int num\_procs;

    MPI\_Comm\_size(MPI\_COMM\_WORLD, &num\_procs);

    int rank;

    MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

    if (rank == 0){

        int n;

        printf("Enter number of elements : ");

        scanf("%d", &n);

        int arr[n];

        for (int i = 0; i < n; i++)

            arr[i] = rand() % 10 + 1;

        printf("Array is -\n [ ");

        for (int i = 0; i < n; i++)

            printf("%d ", arr[i]);

        printf("]\n");

        int elem\_to\_send = n / num\_procs;

        int tag = 0;

        for (int i = 1; i < num\_procs; i++){

            if (i != num\_procs - 1){

                elem\_to\_send = n / num\_procs;

                MPI\_Send(&elem\_to\_send, 1, MPI\_INT, i, i + num\_procs, MPI\_COMM\_WORLD);

MPI\_Send(&arr[i \* (elem\_to\_send)], elem\_to\_send, MPI\_INT, i, i + num\_procs + 1, MPI\_COMM\_WORLD);

continue;

            }

            elem\_to\_send = n / num\_procs + n % num\_procs;

            MPI\_Send(&elem\_to\_send, 1, MPI\_INT, i, i + num\_procs, MPI\_COMM\_WORLD);

MPI\_Send(&arr[(num\_procs - 1) \* (n / num\_procs)], elem\_to\_send, MPI\_INT, i, i + num\_procs + 1, MPI\_COMM\_WORLD);

        }

        int ans = 0;

        for (int i = 0; i < n / num\_procs; i++)

            ans += arr[i];

        int s\_rec;

        for (int i = 1; i < num\_procs; i++){

            s\_rec = 0;

            MPI\_Recv(&s\_rec, 1, MPI\_INT, i, i + num\_procs + 2, MPI\_COMM\_WORLD,

                     MPI\_STATUS\_IGNORE);

            ans += s\_rec;

        }

        printf("Total sum of array is %d\n", ans);

    }

    else{

        int size;

        MPI\_Recv(&size, 1, MPI\_INT, 0, rank + num\_procs, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

        int arr[size];

        MPI\_Recv(arr, size, MPI\_INT, 0, rank + num\_procs + 1, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

        int local = 0;

        for (int i = 0; i < size; i++)

            local = local + arr[i];

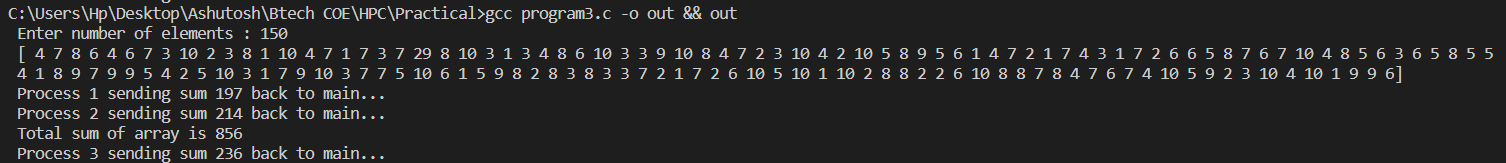
        printf("\nProcess %d sending sum %d back to main...\n", rank, local);

        MPI\_Send(&local, 1, MPI\_INT, 0, rank + num\_procs + 2, MPI\_COMM\_WORLD);

    }

    MPI\_Finalize();

}



**4. Write a program to illustrate basic MPI communication routines**

#include <mpi.h>

#include <stdio.h>

int main(int argc, char \*\*argv){

    MPI\_Init(NULL, NULL);

    int world\_size;

    MPI\_Comm\_size(MPI\_COMM\_WORLD, &world\_size);

    int world\_rank;

    MPI\_Comm\_rank(MPI\_COMM\_WORLD, &world\_rank);

    char processor\_name[MPI\_MAX\_PROCESSOR\_NAME];

    int name\_len;

    MPI\_Get\_processor\_name(processor\_name, &name\_len);

    printf("Hello world from process %s, rank %d out of %d processes\n\n", processor\_name, world\_rank, world\_size);

    if (world\_rank == 0){

        char \*message = "Hello!";

        MPI\_Send(message, 6, MPI\_CHAR, 1, 0, MPI\_COMM\_WORLD);

    }

    else{

        char message[6];

        MPI\_Recv(message, 6, MPI\_CHAR, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

        printf("Message received!\n");

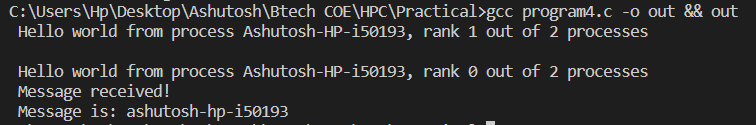
        printf("Message is : %s\n", message);

    }

    MPI\_Finalize();

    return 0;

}



**5. Design a parallel program for summing up an array, matrix multiplication and show logging and tracing MPI activity**

#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

int main(int argc, char \*\*argv){

    MPI\_Init(NULL, NULL);

    int num\_procs;

    MPI\_Comm\_size(MPI\_COMM\_WORLD, &num\_procs);

    int rank;

    MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

    if (rank == 0){

        int n;

        printf("Enter number of elements : ");

        scanf("%d", &n);

        int arr[n];

        for (int i = 0; i < n; i++)

            arr[i] = rand() % 10000 + 1;

        printf("Array is -\n [ ");

        for (int i = 0; i < n; i++)

            printf("%d ", arr[i]);

        printf("]\n");

        int elem\_to\_send = n / 2;

        if (n % 2)

            elem\_to\_send++;

        MPI\_Send(&elem\_to\_send, 1, MPI\_INT, 1, 0, MPI\_COMM\_WORLD);

        MPI\_Send(&arr[n / 2], elem\_to\_send, MPI\_INT, 1, 1, MPI\_COMM\_WORLD);

        float t1 = clock();

        int local = 0;

        for (int i = 0; i < n / 2; i++)

            local = local + arr[i];

        int s\_rec = 0;

        float t2 = clock();

        printf("Time taken by process %d : %f\n", rank, (t2 - t1) / CLOCKS\_PER\_SEC);

        MPI\_Recv(&s\_rec, 1, MPI\_INT, 1, 2, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

        local = local + s\_rec;

        printf("Total sum of array is %d\n", local);

    }

    else{

        float t1 = clock();

        int size;

        MPI\_Recv(&size, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

        int arr[size];

        MPI\_Recv(arr, size, MPI\_INT, 0, 1, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

        float t2 = clock();

        printf("Total time for recieving : %f", (t2 - t1) / CLOCKS\_PER\_SEC);

        t1 = clock();

        int local = 0;

        for (int i = 0; i < size; i++)

            local = local + arr[i];

        printf("\nProcess %d sending sum %d back to main...\n", rank, local);

        t2 = clock();

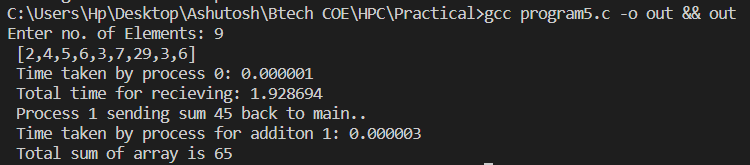
        printf("Time taken by process for addition %d : %f\n", rank, (t2 - t1) / CLOCKS\_PER\_SEC);

        MPI\_Send(&local, 1, MPI\_INT, 0, 2, MPI\_COMM\_WORLD);

    }

    MPI\_Finalize();

}



**6. Write a C program with openMP to implement loop work sharing**

#include <omp.h>

#include <stdio.h>

void reset\_freq(int \*freq, int THREADS){

    for (int i = 0; i < THREADS; i++)

        freq[i] = 0;

}

int main(int \*argc, char \*\*argv){

    int n, THREADS, i;

    printf("Enter the number of iterations :");

    scanf("%d", &n);

    printf("Enter the number of threads (max 8): ");

    scanf("%d", &THREADS);

    int freq[THREADS];

    reset\_freq(freq, THREADS);

    #pragma omp parallel for num\_threads(THREADS)

    for (i = 0; i < n; i++)

        freq[omp\_get\_thread\_num()]++;

    #pragma omp barrier

    printf("\nIn default scheduling, we have the following thread distribution :- \n");

    for (int i = 0; i < THREADS; i++)

        printf("Thread %d : %d iters\n", i, freq[i]);

    int CHUNK;

    printf("\nUsing static scheduling...\n");

    printf("Enter the chunk size :");

    scanf("%d", &CHUNK);

    reset\_freq(freq, THREADS);

    #pragma omp parallel for num\_threads(THREADS) schedule(static, CHUNK)

    for (i = 0; i < n; i++)

        freq[omp\_get\_thread\_num()]++;

    #pragma omp barrier

    printf("\nIn static scheduling, we have the following thread distribution :- \n");

    for (int i = 0; i < THREADS; i++)

        printf("Thread %d : %d iters\n", i, freq[i]);

    printf("\nUsing automatic scheduling...\n");

    reset\_freq(freq, THREADS);

    #pragma omp parallel for num\_threads(THREADS) schedule(auto)

    for (i = 0; i < n; i++)

        freq[omp\_get\_thread\_num()]++;

    #pragma omp barrier

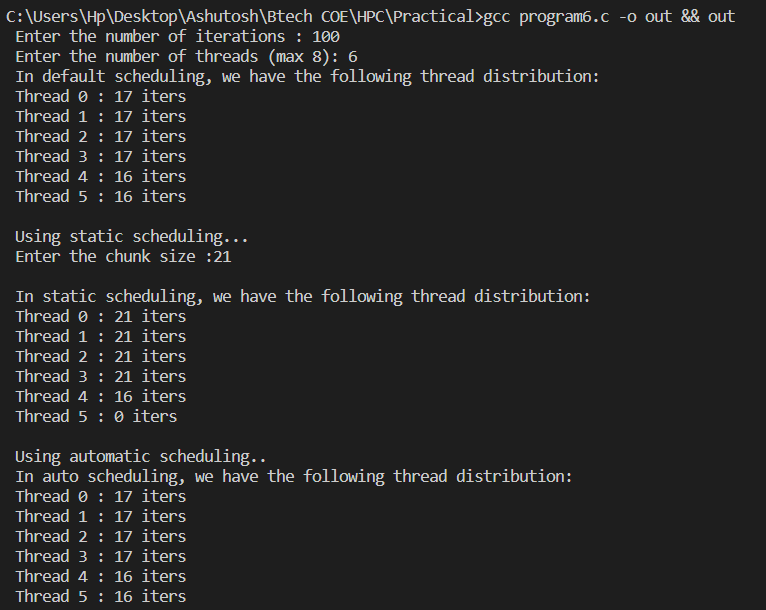
    printf("In auto scheduling, we have the following thread distribution :- \n");

    for (int i = 0; i < THREADS; i++)

        printf("Thread %d : %d iters\n", i, freq[i]);

    return 0;

}



**7. Write a C program with openMP to implement sections work sharing**

#include <omp.h>

#include <stdio.h>

int main(int \*argc, char \*\*argv){

    int num\_threads, THREAD\_COUNT = 4;

    int thread\_ID;

    int section\_sizes[4] = {0, 100, 200, 300};

    printf("Work load sharing of threads...\n");

    #pragma omp parallel private(thread\_ID) num\_threads(THREAD\_COUNT)

    {

        thread\_ID = omp\_get\_thread\_num();

        printf("I am thread number %d!\n", thread\_ID);

        int value\_count = 0;

        if (thread\_ID > 0){

            int work\_load = section\_sizes[thread\_ID];

            for (int i = 0; i < work\_load; i++)

                value\_count++;

            printf("Number of values computed : %d\n", value\_count);

        }

        #pragma omp barrier

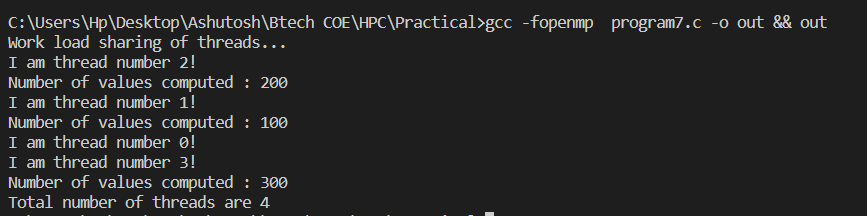
        if (thread\_ID == 0)

            printf("Total number of threads are %d", omp\_get\_num\_threads());

    }

    return 0;

}



**8. Write a program to illustrate process synchronization and collective data movements**

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int thread\_count;

struct arguments{

    int size;

    int \*arr1;

    int \*arr2;

    int \*dot;

};

void \*add\_into\_one(void \*arguments);

void print\_vector(int n, int \*arr){

    printf("[ ");

    for (int i = 0; i < n; i++)

        printf("%d ", arr[i]);

    printf("] \n");

}

int main(int argc, char \*argv[]){

    long thread;

    pthread\_t \*thread\_handles;

    thread\_count = 2;

    thread\_handles = malloc(thread\_count \* sizeof(pthread\_t));

    printf("Enter the size of the vectors : ");

    int n;

    scanf("%d", &n);

    printf("Enter the max\_val of the vectors : ");

    int max\_val;

    scanf("%d", &max\_val);

    struct arguments \*args[2];

    for (int i = 0; i < 2; i++){

        args[i] = malloc(sizeof(struct arguments) \* 1);

        args[i]->size = n;

        args[i]->arr1 = malloc(sizeof(int) \* n);

        args[i]->arr2 = malloc(sizeof(int) \* n);

        args[i]->dot = malloc(sizeof(int) \* n);

        for (int j = 0; j < n; j++){

            args[i]->arr1[j] = rand() % max\_val;

            args[i]->arr2[j] = rand() % max\_val;

        }

    }

    printf("Vectors are : \n");

    print\_vector(n, args[0]->arr1);

    print\_vector(n, args[0]->arr2);

    print\_vector(n, args[1]->arr1);

    print\_vector(n, args[1]->arr2);

    int result[n];

    memset(result, 0, n \* sizeof(int));

    for (thread = 0; thread < thread\_count; thread++){

        printf("Multiplying %ld and %ld with thread %ld...\n", thread + 1, thread + 2, thread);

        pthread\_create(&thread\_handles[thread], NULL, add\_into\_one, (void \*)args[thread]);

    }

    printf("Hello from the main thread\n");

    for (thread = 0; thread < thread\_count; thread++)

        pthread\_join(thread\_handles[thread], NULL);

    for (int i = 0; i < 2; i++){

        printf("Multiplication for vector %d and %d \n", i + 1, i + 2);

        print\_vector(n, args[i]->dot);

        printf("\n");

    }

    free(thread\_handles);

    for (int i = 0; i < n; i++)

        result[i] = args[0]->dot[i] + args[1]->dot[i];

    printf("Result is : \n");

    print\_vector(n, result);

    return 0;

}

void \*add\_into\_one(void \*argument){

    struct arguments \*args = argument;

    int n = args->size;

    for (int i = 0; i < n; i++)

        args->dot[i] = args->arr1[i] \* args->arr2[i];

    return NULL;

}

